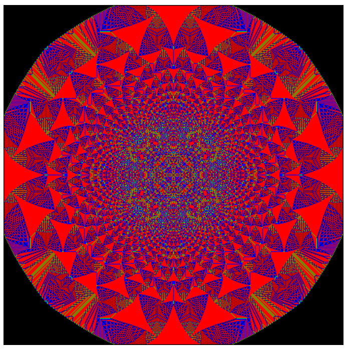


Figure 1: Final stable state of a 513x513 Abelian sandpile initialised with the center cell set to 526338 and the rest 0.

CSC2002S
PCP1 Assignment 2024
Parallel Programming with Java: *Parallelizing an Abelian Sandpile Simulation, version 2.*

set by M. M. Kuttel[1]

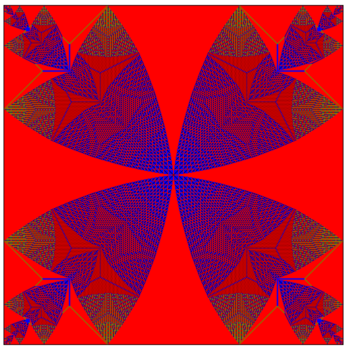


Figure 2: Final stable state of a 513x513 Abelian sandpile initialised with all cells set to 4.

1. Introduction

This first PCP assignment will give you experience in programming a parallel algorithm for the shared memory parallel programming model using Java. The algorithm that you will parallelise runs a simulation of a *cellular automaton*. A cellular automaton comprises a grid of cells of specific integer values. These cells evolve according to a set of *rules* based on the states of their neighboring cells. The rules are applied over a series of discrete time steps. Cellular automata demonstrate how simple rules can lead to complex, emergent behaviors. They have been studied as models of biological systems, the best known of which is the two-dimensional Conway's Game of Life[2]. However, this assignment is concerned with a variety of two-dimensional cellular automaton called an **Abelian Sandpile**.

In the Abelian Sandpile, each cell contains a certain number of "grains of sand". When the number of grains in a cell exceeds 4, in the next time step the cell "topples", distributing grains evenly to its neighboring cells (left, right, up and down) and keeping any remainder. The edge, or border, of the automaton grid is a "sink" – for cells on the border, their distribution to the border will "disappear" into the sink. An Abelian Sandpile evolves from a starting configuration (usually something simple like all cells set to the value 4) until it reaches a *stable state,* where all cells have fewer than 4 grains. If this final state is coloured according to the cell values (black for 0, green for 1, blue for 2 and red for 3) is often revealed to be a beautiful pattern with complex symmetry. Two examples of the final state of 512x512 grids are shown in Figures 1 and 2 above. As the final stable configuration is unique regardless of the order in which cell distributions are computed[3], calculation of the change in the sandpiles in each successive timestep for a simulation may be parallelized fairly simply. This is your task in this assignment: starting from a given simple serial program, you will code a parallel version, validate it against the serial solution for correctness and then benchmark it against the serial algorithm to determine under which conditions (if any) the parallelization was worth the extra programming effort required.

2. The serial solution

You are provided with a Java program that codes a serial solution to the simulation of the cellular automaton. The program hard codes the rules for the Abelian Sandpile that we will use for this assignment, but is generally applicable to other similar automata. Your first step should be to have a good look at the serial code. You will not and cannot change this code.

[1] Adapted from "ABELIAN SANDPILE SIMULATION", Peachy Parallel Assignments (EduPar 2022), 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW

[2] https://mathworld.wolfram.com/GameofLife.html

[3] Deepak Dhar. *Self-organized critical state of sandpile automaton models*. Phys. Rev. Lett., 64:1613–1616, Apr 1990.

2.1 INPUT

The program takes the following **command line arguments** in order:

input file name, output file name.

The input file is in comma separated value format (.csv), where the first line contains the number of rows and number of columns in the data. The following lines are the rows in the file, with values for each column in the row. There are some example files provided in the **input** folder, but you can (and should) create more for testing. There are no checks for correct format of the file – you may add some to your parallel solution.

2.2 JAVA CLASSES

There are two classes in this the simple program, as follows. The AutomatonSimulation class runs the whole simulation; the Grid class represents the cellular automaton.

2.3 DEBUG MODE

If the DEBUG flag in the code is set to true (you have to change the code to do this), the program will output text representations of the grid at each step. It is not advisable to do this for large files…

2.4 RUNNING THE CODE

Once you have had a look at the code, run it to see how it works. Experiment with different inputs with differing numbers of rows and columns and cell values.

The code contains a **makefile** in the top folder, to make execution on a Linux system straightforward. If you do not know what a makefile is, look it up and have a look at the example provided.

To run the code on a Linux (or MacOS) machine, copy the zip file onto the computer, unzip it:

```
unzip PCP_ParallelAssignment2024.zip
```

change into the top folder:

```
cd PCP_ParallelAssignment2024/
```

and then execute the command:

```
make run
```

This will execute the code with the default input parameters `input/65_by_65_all_4.csv output/65_by_65_all_4.png`

If you would like to run with other parameters, then execute providing the other parameters, e.g:

```
make run ARGS="input/517_by_517_centre_534578.csv output/517.png"
```

Note that these instructions are a guide – it is your task to figure out how to execute the code provided (i.e. do NOT ask the lecturer questions about this).

2.5 OUTPUT

After simulations has completed, the program outputs to the screen the number of rows and columns, the number of simulation timesteps required to reach the stable state and the total time taken. A Portable Networks Graphic (.png) file is also written of the final stable state.

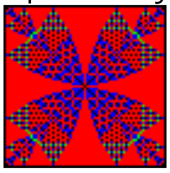
For example, if the program is executed as follows:

```
java -classpath bin serialAbelianSandpile.AutomatonSimulation
input/65_by_65_all_4.csv output/65_by_65_all_4.png
```

The output to the screen will be (the time will vary depending on the computer on which you run it):

```
Rows: 65, Columns: 65
Simulation complete, writing image...
Number of steps to stable state: 1156
Time: 27 ms
```

And `output/65_by_65_all_4.png` will look like this:



3. Your parallel solution

Your task in this assignment is to create a parallel version of the serial algorithm that is correct and runs faster. The parallel solution must avoid data races, by ensuring that threads compute non-overlapping parts of the grid and that threads only proceed to the next timestep once all are done. You can only use the **join()** synchronisation mechanism in the assignment – no other synchronisation mechanisms are allowed (e.g. no barriers, synchonised methods, no atomic variables etc. Note that parallel programs need to be **both** *correct* and *faster* than the serial versions. Therefore, you need to demonstrate both correctness and speedup for your assignment. If speedup over the serial solution is not achieved, you need to explain why this happens.

3.1 Specific requirements

In this assignment, you must do the following.

1. Profile the **serial program,** measuring the time taken to run for a range of input sizes and starting values for the cells..
2. Write **parallel version** using the Java Fork/Join framework in order to speed up the algorithm. This version **must have the same output text and file output as the serial solution** (although you can output other files as well, if you choose).
3. **Validate** your parallel version to demonstrate that it works correctly (i.e. produces the same solution as the serial version).
4. **Benchmark** your parallel program experimentally, **timing** the **execution** on at *least two* machines (e.g. your laptop and the departmental server) with **different input sizes**, generating **speedup graphs** that show the conditions under which you get the best parallel **speedup**. (Do *not* give timing graphs!)
5. Write a **short report** including the graphs with an explanation of your findings.

4. Report

You must submit a short assignment report **in pdf format** (may not submit Word documents). Your **concise** report must contain the following:

A **brief** *Methods* section describing:

- your **parallelization approach,** including any particular issues/considerations and **optimisations** you implemented, and how you determined a **good sequential cutoff**;

- how you **validated** your algorithms (showed that it is correctly implemented);
- how you **benchmarked** your algorithm,
- the **machine architectures** you tested on; and
- any **problems/difficulties** you encountered.

A *Results* section, with **speedup graphs**.

- Graphs must show how the parallel algorithm scales with **grid size** and number of **searches**, and on (at least 2) different computers, one of which needs to be a mulitcore machine like the departmental server. **Graphs** should be clear and **labelled** (title and axes).
- **A brief discussion** that answers following questions must be included. This should answer the following questions.
    - For what range of grid sizes does your parallel program perform well?
    - What is the maximum speedup you obtained and how close is this speedup to the ideal expected?
    - How reliable are your measurements?
    - Are they any anomalies and can you explain why they occur?

A *Conclusions* (note the plural) section where you say whether it is worth using parallelization (multithreading) to tackle this problem in Java.

The report must be no more than five pages, including graphs. To do this, have multiple graphs on a single axis.

5. Assignment submission requirements

You will need to submit your assignment in two places:

1. Submit the **code** to the automarker as a zipped **archive.** Your submission archive must contain:
    - **all the files** needed to run your solution
    - a **Makefile** for compilation **that works as the serial solution does**, e.g. the following command must run your parallel solution correctly.
      ```
      make run ARGS="input/517_by_517_centre_534578.csv output/517.png"
      ```
    - **a GIT usage log (**as a .txt file, use `git log –all` to display all commits and save).

2. Submit your report (**pdf format only**) to the Amathuba assignment.

It is your responsibility to check that all uploaded files are correct.
Remember that the deadline for marking **queries** on your assignment is **one week after the return of your mark.** After this time, you may not query your mark.

1.6 Assignment marking

A draft marking guide is included below.

CSC2002S PCP1 2024 PARALLEL ASSIGNMENT	Max Mark
CODE	
Code uses the using fork/join framework correctly. There must be actual calls to fork and join in the right order, and a variable sequential cutoff.	*2*
Good clear code **style and organization**. Efficient algorithms with no odd/unnecessary code. No obvious errors or obvious inefficiencies .	*2*
Code comments are clear and informative and placed where needed.	*1*
REPORT: INTRODUCTION AND METHODS	
Parallelisation approach clearly explained in sufficient detail: how the fork/join framework was used, how the divide-and-conquer parallel algorithm works. It must explain in sufficient detail to give a clear general idea of how the parallelization was done. **The approach explained must match the actual code submitted.**	*2*
Validation of the algorithm (checks of correctness) clearly described. Claims of validation must match what actually happens with the code.	*2*
Benchmarking correctly performed and clearly explained in sufficient detail - at least 2 machine architectures, one of which is >4 cores, good range of data sizes. Sufficient detail provided so that someone else could do the same thing.	*2*
REPORT: DISCUSSION	
Speedup Graphs: Graphs of parallel speedup versus grid size on two different architectures. NOTE THAT THESE MUST BE SPEEDUP GRAPHS - no marks for timing graphs! The graphs must match the code submitted.	*2*
Graph presentation/format/design: clear and easy to understand/comprehend. All axes labelled.	*2*
Discussion of the range of data sizes for which the best speedup is obtained, and why, well justified. Should explain where and why the best speedup is obtained.	*2*
Discussion of how the maximum speedup compares to the ideal for each architecture.	*2*
Trends and any anomalies (spikes etc.) in the graphs discussed and explained.	*2*
CONCLUSIONS	
Was parallelization worth it? - conclusions justified and explained. The conclusions make sense in the context of the results presented in the report.	*2*
Clarity of the discussion - overall how clear and understandable is the report, especially the presentation and discussion of the results?	*2*
TOTAL	***25***
PENALTIES	
Parallel code does not execute/run with basic make instruction/no Makefile.	**-5**
No GIT repository.	**-2**
GIT repository, but no regular updates (commits on at least on 3 separate days).	**-1**
Late penalty (10% for first day or part thereof). No late handins after one day late!	***-2.5***